2011 Latin-American Symposium on Dependable Computing

Timing Analysis of Leader-based and Decentralized Byzantine Consensus Algorithms

Fatemeh Borran, Martin Hutle, and André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
{*fatemeh.borran, martin.hutle, andre.schiper*}*@epfl.ch*

***Abstract*—We compare in an analytical way two leader-based and decentralized algorithms (that is, algorithms that do not use a leader) for Byzantine consensus with strong validity. We show that for *the algorithms we analyzed*, in most cases, the decentralized variant of the algorithm shows a better worst-case execution time. Moreover, for the practically relevant case $t \leq 2$ (t is the maximum number of Byzantine processes), this worst-case execution time is even at least as good as the execution time of the leader-based algorithms in fault-free runs.**

***Keywords*-distributed algorithms, Byzantine consensus, timing analysis**

I. Introduction

Consensus is a fundamental building block for fault-tolerant distributed systems. Algorithms for solving the consensus problem can be classified into two broad categories: *leader-based* algorithms, that use the notion of a (changing) leader, and *decentralized* algorithms, where no such dedicated process is used. Most of the consensus algorithms proposed in early 80's, for both synchronous and asynchronous systems,[1] are decentralized (e.g., [1], [2], [3], [4]). Later a leader (or coordinator) was introduced, in order to reduce the message complexity and/or improve the best case performance (e.g., [5], [6], [7]). However, recently it has been pointed out that the leader-based PBFT Byzantine consensus algorithm [8], which assumes a partially synchronous system [5], is vulnerable to performance degradation [9], [10]. According to these two papers, a malicious leader can introduce latency into the global communication path simply by delaying the message that it has to send. Moreover, a malicious leader can manipulate the protocol timeout and slow down the system throughput without being detected. This motivated the development of decentralized Byzantine consensus algorithms for partial synchronous systems [11]. The next step, addressed here, is to compare the theoretical execution time of decentralized and leader-based consensus algorithms. We study the question analytically in the model considered in [8] for PBFT, namely a partially synchronous system in which the end-to-end messages transmission delay δ is unknown.

[1] In asynchronous systems, using randomization to solve probabilistic consensus.

Our paper analyzes two Byzantine consensus algorithms for strong validity, each one with a decentralized and a leader-based variant. One of these two algorithms is inspired by *Fast Byzantine Paxos* [12], the other by *PBFT*. Our analysis shows the superiority of the decentralized variants over the leader-based variants. First, the analysis shows that for the decentralized variants the worst case performance and the fault-free case performance overlap, which is not the case for the leader-based variants. Second, it shows that the worst case of the decentralized variant of our two algorithms is always better than the worst case of its leader-based variant. Third, for $t \leq 2$ (t is the maximum number of Byzantine processes), it shows that the worst case execution time of our decentralized variant is never worse than the execution time of the leader-based variant in fault-free runs. As future work, we plan to extend our study to consensus algorithms with weak validity, as Fast Byzantine Paxos and PBFT.

In the next section we give the system model for our analysis and introduce the round model we use for the description of our algorithms. Section III presents in a modular way the consensus algorithms under consideration. In Section IV, we give the implementation of the round model. Section V contains our main contribution, the analysis and comparison of the algorithms.

II. Definitions and System Model

A. System Model

We consider a set Π of n processes, among which at most t can be faulty. A faulty process behaves arbitrarily. Non-faulty processes are called *correct* processes, and $\mathcal{C}$ denotes the set of correct processes.

Processes communicate through message passing, and the system is partially synchronous [5]. Instead of separate bounds on the process speeds and the transmission delay, we assume that in every run there is a bound δ on the *end-to-end transmission delay* between correct processes, that is, the time between the sending of a message and the time where this message is *actually received* (this incorporates the time for the transmission of the message and of possibly several steps until the process makes a receive step that includes this message). This is the same model considered in [8] for PBFT. We do not make use of digital signatures.

978-0-7695-4320-8/11 $26.00 © 2011 IEEE
DOI 10.1109/LADC.2011.12

However, the communication channels are authenticated, i.e., the receiver of a message knows the identity of the sender. In addition, we assume that processes have access to a local non-synchronized clock; for simplicity we assume that this clock is drift-free.

B. Round Model

As in [5], we consider rounds on top of the system model. This improves the clarity of the algorithms, makes it simpler to change implementation options, and makes the timing analysis easier to understand. In the round model, processing is divided into rounds of message exchange.

In each round r, a process p sends a message according to a sending function S_p^r to a subset of processes and, at the end of this round, computes a new state according to a transition function T_p^r, based on the messages it received and its current state. Note that this implies that a message sent in round r can only be received in round r (rounds are *communication closed*). The message sent by a correct process is denoted by σ_p^r; messages received by process p in round r are denoted by $\vec{\mu}_p^r$ ($\vec{\mu}_p^r$ is a vector, with one entry per process; $\vec{\mu}_p^r[q] = \perp$ means that p received no message from q). In all rounds, we assume the following *integrity* predicate $\mathcal{P}_{int}(r)$, which states that if a correct process p receives a message from a correct process q, then this message was sent by q:

$$\mathcal{P}_{int}(r) \equiv \forall p, q \in \mathcal{C} : \ \vec{\mu}_p^r[q] \in \{\perp, \sigma_q^r\}$$

In a partially synchronous system it is possible to ensure the following property: there exists some round GSR (*Global Stabilization Round*) such that for all rounds $r \geq GSR$, the message sent in round r by a correct process q to a correct process p is received by p in round r. This is expressed by $\forall r \geq GSR : \mathcal{P}_{sync}(r)$, where

$$\mathcal{P}_{sync}(r) \equiv \forall p, q \in \mathcal{C} : \ \vec{\mu}_p^r[q] = \sigma_q^r$$

We say that such a round r is *synchronous*. We further need the definition of a *consistent* round. In such a round, correct processes receive the same set of messages:

$$\mathcal{P}_{cons}(r) \equiv \forall p, q \in \mathcal{C} : \ \vec{\mu}_p^r = \vec{\mu}_q^r$$

Consensus algorithms consist of a sequence of phases, where each phase consists of one or more rounds. For our consensus algorithms, we need eventually a phase where all rounds are synchronous, and the first round is consistent. A round in which $\mathcal{P}_{cons}$ eventually holds will be called a *WIC round* (Weak Interactive Consistency, defined in [13]). Eventually synchronous rounds are provided by the implementation of the round model, which is discussed in Section IV. Ensuring eventually consistent rounds can by done in a leader-based or decentralized way, and discussed in Section III.

Algorithm 1 MA algorithm with $n > 5t$ (code of process p) [11], [13]

```
1:  State:
2:    x_p ∈ V
3:    decision_p ∈ V

4:  Round r = 2φ − 1 :                                   /* WIC round */
5:    S_p^r:
6:      send ⟨x_p⟩ to all processes
7:    T_p^r:
8:      if number of non-⊥ elements in μ⃗_p^r ≥ n − t then
9:        x_p ← smallest most frequent non-⊥ element in μ⃗_p^r

10: Round r = 2φ :
11:   S_p^r:
12:     send ⟨x_p⟩ to all processes
13:   T_p^r:
14:     if n − t elements in μ⃗_p^r are equal to v ≠⊥ then
15:       decision_p ← v
```

C. Byzantine Consensus

In the consensus problem each process has an initial value, and processes must decide on one value that satisfies the following properties:

- *Strong validity:* If all correct processes have the same initial value, this is the only possible decision value.
- *Agreement:* No two correct processes decide differently.
- *Termination:* All correct processes eventually decide.

In the paper we analyze a sequence of consensus instances.

III. Consensus Algorithms

In this section we present the two consensus algorithms, both from [11], [13], that we use for our analysis. Both require a round in which $\mathcal{P}_{cons}$ eventually holds. Then we give two implementations of WIC rounds, one leader-based, the other decentralized. By combining the two consensus algorithms with the two WIC implementations we get four algorithms that will be analyzed.

A. Consensus algorithms with WIC rounds

1) The MA algorithm: The MA algorithm [11], [13] (Algorithm 1) is inspired by the FaB Paxos algorithm proposed by Martin and Alvisi [12].[2] A phase of Algorithm 1 consists of two rounds. The algorithm is safe with $t < n/5$. For termination, the two rounds of a phase must eventually be synchronous, and the first round must be a WIC round.

Agreement follows from the fact that once a process decided, at least $n - 2t$ correct processes have the same estimate x, and thus no other value will ever be adopted in line 9. A similar argument is used for validity. Termination follows from the fact that in a round $2\phi - 1 \geq GSR$ with a consistent reception vector $\vec{\mu}_p^r$ all correct processes adopt

[2]FaB Paxos is expressed using "proposers", "acceptors" and "learners". MA is expressed without these roles. Moreover, FaB Paxos solves consensus with *weak validity*, while MA solves consensus with *strong validity*. In addition, MA is expressed using rounds.

Algorithm 2 Leader-based implementation of a WIC round with $n > 3t$ (code of process p) [13]

```
1: Initialization:
2:   ∀q ∈ Π : received_p[q] ← ⊥

3: Round ρ = 1 :
4:   S_p^ρ:
5:     send ⟨m_p⟩ to all
6:   T_p^ρ:
7:     received_p ← μ⃗_p^ρ

8: Round ρ = 2 :
9:   S_p^ρ:
10:    send ⟨received_p⟩ to coord_p
11:  T_p^ρ:
12:    if p = coord_p then
13:      for all q ∈ Π do
14:        if |{q' ∈ Π : μ⃗_p^ρ[q'][q] = received_p[q]}| < 2t + 1 then
15:          received_p[q] ← ⊥

16: Round ρ = 3 :
17:  S_p^ρ:
18:    send ⟨received_p⟩ to all
19:  T_p^ρ:
20:    for all q ∈ Π do
21:      if (μ⃗_p^ρ[coord_p][q] ≠ ⊥) ∧ |{i ∈ Π : μ⃗_p^ρ[i][q] =
         μ⃗_p^ρ[coord_p][q]}| ≥ t + 1 then
22:        M⃗_p[q] ← μ⃗_p^ρ[coord_p][q]
23:      else
24:        M⃗_p[q] ← ⊥
```

the same value in line 9, and thus decide on this value in round 2ϕ.

2) The CL algorithm: The CL algorithm [11], [13] is inspired by the PBFT algorithm proposed by Castro and Liskov [8], expressed using rounds, including one WIC round.[3] A phase consists of three rounds. The algorithm is safe with $t < n/3$. For termination, the three rounds of a phase must eventually be synchronous and the first round must be a WIC round. The detailed code of the CL algorithm is not essential for understanding the rest of the paper. Actually, the analysis is the same for any algorithm that requires three rounds per phase, with a first WIC round.

B. Implementation of a WIC round

We consider two implementations for a WIC round: one leader-based and one decentralized. The implementations are also expressed using rounds, in order to distinguish them from the "normal" rounds, we use ρ to denote these rounds. The implementation has to be understood as follows. Let r be a WIC round, e.g., round $r = 2\phi - 1$ of Algorithm 1. The messages sent in round $r = 2\phi - 1$ are used as the input variable m_p in the WIC implementation (see Algorithm 2). The resulting vector provided by the WIC implementation, denoted by $\vec{M}_p$ (Algorithm 2) is then passed to the transition function of round r as the reception vector, $\vec{\mu}_p^r$.

[3]PBFT solves a sequence of consensus instances with *weak validity*, while CL solves consensus with *strong validity*.

Algorithm 3 Decentralized implementation of a WIC round with $n > 3t$ (code of process p) [11]

```
1: Initialization:
2:   W_p ← {⟨λ, m_p⟩}

3: Round ρ, 1 ≤ ρ ≤ t + 1 :
4:   S_p^ρ:
5:     send {⟨α, v⟩ ∈ W_p : |α| = ρ − 1 ∧ p ∉ α ∧ v ≠ ⊥} to all
6:   T_p^ρ:
7:     for all {q | ⟨α, v⟩ ∈ W_p ∧ |α| = ρ − 1 ∧ q ∈ Π ∧ q ∉ α} do
8:       if ⟨β, v⟩ is received from process q then
9:         W_p ← W_p ∪ {⟨βq, v⟩}
10:      else
11:        W_p ← W_p ∪ {⟨βq, ⊥⟩}
12:    if ρ = t + 1 then
13:      for all ⟨α, v⟩ ∈ W_p from |α| = t to |α| = 1 do
14:        W_p ← W_p \ ⟨α, v⟩
15:        if ∃v' s.t. |⟨αq, v'⟩ ∈ W_p| ≥ n − |α| − t then
16:          W_p ← W_p ∪ ⟨α, v'⟩
17:        else
18:          W_p ← W_p ∪ ⟨α, ⊥⟩
19:      for all q ∈ Π do
20:        M⃗_p[q] ← v  s.t. ⟨q, v⟩ ∈ W_p
```

1) Leader-based implementation: Algorithm 2, which appears in [13], implements WIC rounds using a leader. If a correct process is the coordinator, all processes receive the same set of messages from this process in round $\rho = 3$.

In round $\rho = 2$, the coordinator compares the value received from some process p with the value indirectly received from other processes. If at least $2t + 1$ same values have been received, the coordinator keeps that value, otherwise it sets the value to $\perp$. This guarantees that if the coordinator keeps v, at least $t + 1$ correct processes have received v from p in round $\rho = 1$. Finally, in round $\rho = 3$ every process sends values received in round $\rho = 1$ or $\perp$ to all. Each process verifies whether at least $t+1$ processes validate the value that it has received from the coordinator in round $\rho = 3$. Rounds $\rho = 1$ and $\rho = 3$ are used to verify that a faulty leader cannot forge the message from another process (integrity).

Since a WIC round can be ensured only with a correct coordinator, we need to ensure that the coordinator is eventually correct. In Section IV we do so by using a *rotating coordinator*. A WIC round using this leader-based implementation needs three "normal" rounds.

2) Decentralized implementation: Algorithm 3 is a decentralized (without leader) implementation of a WIC round [11]. It is based on the *Exponential Information Gathering* (EIG) algorithm for synchronous systems proposed by Pease, Shostak and Lamport [1]. Initially, process p has its initial value m_p given by round $r = 2\phi - 1$ of the consensus algorithm (e.g., Algorithm 1). Throughout the execution, processes learn about the initial values of other processes. The information can be organized inside a tree. Each node of the tree constructed by process p has a label and a value. The root has an empty label λ and a value m_p. Process p maintains the tree using a set W_p. When p receives a

message $\langle \beta, v \rangle$ from q adds $\langle \beta q, v \rangle$ to W_p, otherwise it adds $\langle \beta q, \perp \rangle$. After $t+1$ rounds, badly-formatted messages in W_p are dropped, and all correct processes have the same value for W_p.

Similarly to the leader-based implementation, it requires $n > 3t$. On the other hand, a WIC round using this decentralized implementation needs $t + 1$ "normal" rounds.

C. The four combinations

Combining the two WIC based algorithms, namely MA and CL, with the two implementations of WIC rounds, namely leader-based (L) and decentralized (D), we get four algorithms, denoted by MA-L, MA-D, CL-L and CL-D. Phases have the following lengths: four rounds for MA-L, $t+2$ rounds for MA-D, five rounds for CL-L and $t+3$ rounds for CL-D.

IV. Round Implementation

As already mentioned in Section II-A, we consider a partially synchronous system with an unknown bound δ on the end-to-end transmission delay between correct processes. The main technique to find the unknown δ in the literature is using an adaptive timeout, i.e., starting the first phase of an algorithm with a small timeout Γ_0 and increasing it from time to time. The timeout required for an algorithm can be calculated based on the bound δ and the number of rounds needed by one phase of the algorithm. The approach proposed in the DLS model [5] is to increase the timeout linearly, while recent works, e.g., PBFT [8], increase the timeout exponentially.

The main question is when to increase the timeout? Increasing the timeout in every phase provides a simple solution, in which all processes adapt the same timeout for a given phase. However, this is not an efficient solution, since processes might increase the timeout unnecessarily. An efficient solution is increasing the timeout when a correct process requires that. This occurs typically when a correct process is unable to terminate the algorithm with the current timeout. The problem with this solution is that different processes might increase the timeout at different points in time.

For leader-based algorithms, a related question is the relationship between leader change and timeout change. Most of the existing protocols apply both timeout and leader modifications at the same time [5], [8], [12], [14], [9], [10]. Our round implementation allows decoupling timeout modification and leader modification. We show that such a strategy performs better than the traditional strategies in the worst case.

A. The algorithm

Algorithm 4 describes the round implementation. The main idea of the algorithm is to synchronize processes to the same round (round synchronization). The algorithm requires view synchronization (eventually processes are in the same view) in addition to the round synchronization. This is because processes might increase the timeout at different rounds. The view number is thus used to synchronize the processes' timeout.

Each process p keeps a round number r_p and a view number v_p, initially equal to 1. While the round number corresponds also to the round number of the consensus algorithm, the view number increases only upon reconfiguration. Thus, the leader and the timeout are functions of the view number. The leader changes whenever the view changes, based on the rotating leader paradigm (line 7). Note that the value of $coord_p$ is ignored in decentralized algorithms. The timeout does not necessarily change whenever the view changes. After line 7 a process starts the *input & send* part, in which it queries the input queue for new proposals (using a function $input()$, line 8), initializes new slots on the $state$ vector for each new proposal (line 10), calls the send function of all active consensus instances (line 13), and sends the resulting messages (line 16). The process then sets a timeout for the current round using a deterministic function Γ based on its view number v_p (line 17), and starts the *receive* part, where it collects messages (line 22). Basically, this part uses an init/echo message scheme for round synchronization based on ideas that appear already in [15], [5], [16]. The receive part is described later. Next, in the *comp. & output* part, the process calls the state transition function of each active instance (line 41), and outputs any new decisions (line 44) using the function $output()$. Finally, a check is done at the end of each phase, i.e., only if $next\_r_p \bmod \alpha = 1$ (line 45), where α represents the number of rounds in a phase . The check may lead to request a view change, therefore, the check is skipped if $v_p \neq next\_v_p$ (the view changes anyway). The check is whether all instances, started at the beginning of the phase, have decided (lines 45-46). If not, the process concludes that the current view was not successful (either the current timeout was small or the coordinator was faulty), and it expresses its intention to start the next view by sending an INIT message for view $v_p + 1$ (line 47).

The function $init(v)$ (line 10) gives the initial state for initial value v of the consensus algorithm; respectively, $decision(state)$ (line 42) gives the decision value of the current state of the consensus algorithm, or $\perp$ if the process has not yet decided.

Receive part: To prevent a Byzantine process from increasing the round number and view number unnecessarily, the algorithm uses two different type of messages, INIT messages and START messages. Process p expresses the intention to enter a new round r or new view v by sending an INIT message. For instance, when the timeout for the current round expires, the process — instead of starting immediately the next round — sends an INIT message (line 20) and waits

Algorithm 4 A round implementation for Byzantine faults with $n > 3t$ (code of process p)

```
1:  r_p ← 1; next_r_p ← 1                                   /* round number */
2:  Rcv_p ← ∅                                               /* set of received messages */
3:  ∀i ∈ ℕ : state_p[i] ← ⊥                                 /* state of instance i */
4:  ∀i ∈ ℕ : start_p[i] ← 0                                 /* starting round for instance i */
5:  v_p ← 1; next_v_p ← 1                                   /* view number */

6:  while true do
7:    coord_p ← p_((v_p − 1 mod n)+1)
                                                                              input & send
8:    I ← input()
9:    for all ⟨i, v⟩ ∈ I do
10:     state_p[i] ← init(v)
                  /* initialization of state with initial value v */
11:     start_p[i] ← r_p
12:   for all i : state_p[i] ≠ ⊥ do
13:     msgs[i] ← S_p^{r_p}(state_p[i], coord_p)
14:   for all q ∈ Π do
15:     M_q ← {⟨i, msgs[i][q]⟩ : state_p[i] ≠ ⊥ }
16:     send(START, M_q, v_p, r_p, p) to q
                                                                              receive
17:   timeout_p ← current_time + Γ(v_p)
18:   while next_v_p = v_p and next_r_p = r_p do
19:     if current_time ≥ timeout_p then
20:       send(INIT, v_p, r_p + 1, p) to all
21:     receive(M)
22:     Rcv_p ← Rcv_p ∪ M
23:     if ∃ r and t + 1 q s.t. ⟨INIT, v_p, r + 1, q⟩ ∈ Rcv_p then
24:       let r_0 be the largest such r
25:       if r_0 ≥ r_p then
26:         next_r_p ← r_0
27:         send(INIT, v_p, r_0 + 1, p) to all
28:     if ∃ v and t + 1 q s.t. ⟨INIT, v + 1, −, q⟩ ∈ Rcv_p then
29:       let v_0 be the largest such v
30:       if v_0 ≥ v_p then
31:         next_v_p ← v_0
32:         send(INIT, v_0 + 1, r_p, p) to all
33:     if ∃ 2t + 1 q s.t. ⟨INIT, v_p, r_p + 1, q⟩ ∈ Rcv_p then
34:       next_r_p ← max{r_p + 1, next_r_p}
35:     if ∃ 2t + 1 q s.t. ⟨INIT, v_p + 1, −, q⟩ ∈ Rcv_p then
36:       next_v_p ← max{v_p + 1, next_v_p}
                                                                              comp. & output
37:   O ← ∅
38:   for all i : state_p[i] ≠ ⊥ do
39:     for all r ∈ [r_p, next_r_p − 1] do
40:       ∀q ∈ Π : M_r[q] ← m
            if ∃M ⟨START, M, v_p, r, q⟩ ∈ Rcv_p ∧ ⟨i, m⟩ ∈ M
            else M_r[q] ← ⊥
41:       state_p[i] ← T_p^r(M_r, state_p[i], coord_p)
42:     if ∃v s.t. decision(state_p[i]) = v for the first time then
43:       O ← O ∪ ⟨i, v⟩     /* v is the decision of instance i */
44:   output(O)
45:   if v_p = next_v_p ∧ next_r_p mod α = 1 then
46:     if ∃i : start_p[i] ≤ next_r_p − α ∧ decision(state_p[i]) = ⊥
        then
47:       send(INIT, v_p + 1, next_r_p, p) to all
48:   r_p ← next_r_p
49:   v_p ← next_v_p
```

that enough processes timeout. If process p in round r_p and view v_p receives at least $2t + 1$ INIT messages for round $r_p + 1$ (line 33), resp. view $v_p + 1$ (line 35), it advances to round $r_p + 1$, resp. to view $v_p + 1$, and sends an START message with current round and view (line 16). If the process receives $t + 1$ INIT messages for round $r + 1$ with $r \geq r_p$, it enters immediately round r (line 23), and sends an INIT message for round $r + 1$. In a similar way, if the process receives $t + 1$ INIT messages for view $v + 1$ with $v \geq v_p$, it enters immediately view v (line 28), and sends an INIT message for view $v + 1$.

Properties of Algorithm 4: The correctness proofs of Algorithm 4 are given in Section IV-D. Here we give the main properties of the algorithm:

1) If one correct process starts round r (resp. view v), then there is at least one correct process that wants to start round r (resp. view v). This is because at most t processes are faulty (see Lemma 1).
2) If all correct processes want to start round $r+1$ (resp. view $v+1$), then all correct processes eventually start round $r+1$ (resp. view $v+1$). This is because $n-t \geq 2t+1$ and lines 33-36 (see Lemma 2).
3) If one correct process starts round r (resp. view v), then all correct processes eventually start round r (resp. view v). This is because a correct process starts round r (resp. view v) if it receives $2t + 1$ INIT messages for round r (resp. view v). Any other correct process in round $r' < r$ (resp. view $v' < v$) will receive at least $t+1$ INIT messages for round r (resp. view v). By lines 23 to 26, these correct processes will start round $r - 1$ (resp. view $v - 1$) and will send an INIT message for round r (resp. view v), see line 27. From item 2, all correct processes eventually start round r (resp. view v). The complete proof is given by Lemmas 3-5.

B. Timing properties of Algorithm 4

Algorithm 4 ensures the following timing properties:

1) *If process p starts round r (resp. view v) at time τ, all correct processes will start round r (resp. view v) by time $\tau+2\delta$.* This is because p has received $2t+1$ INIT messages for round r (resp. view v), at time τ. All correct processes receive at least $t + 1$ INIT messages by time $\tau + \delta$, start round $r - 1$ (resp. view $v - 1$) and send an INIT message for round r (resp. view v). This message takes at most δ time to be received by all correct processes. Therefore, all correct processes receive at least $2t + 1$ INIT messages by time $\tau + 2\delta$, and start round r (resp. view v). The complete proof is given by Lemma 5.
2) *If a correct process p starts round r (view v) at time τ, it will start round $r+1$ the latest by time $\tau+3\delta+\Gamma(v)$.* By item 1, all correct processes start round r, by time $\tau + 2\delta$. Then they wait for the timeout of round r, which is $\Gamma(v)$. Therefore, by time $\tau + 2\delta + \Gamma(v)$ all correct processes timeout for round r, and send an INIT message for round $r + 1$, which takes δ time to be received by all correct processes. Finally, the latest by time $\tau + 3\delta + \Gamma(v)$, process p receives $2t + 1$ INIT messages for round $r + 1$ and starts round $r + 1$. The complete proof is given by Lemma 6.

We can make the following additional observation:

3) *A timeout $\Gamma(v) \geq 3\delta$ for round r (view v) ensures that if a correct process starts round r at time τ, it receives all round r messages from all correct processes before the expiration of the timeout (at time $\tau + 3\delta$).* By item 1, all correct processes start round r, by time $\tau + 2\delta$. The message of round r takes an additional δ time. Therefore a timeout of at least 3δ ensures the stated property. The complete proof is given by Lemma 7.

C. Parameterizations of Algorithm 4

We now discuss different adaptive strategies for the timeout value $\Gamma(v_p)$. First we consider the approach of [5]: increasing the timeout linearly (whenever the view changes). We will refer to this parameterizations by A. Then we consider the approach used by PBFT [8]: increasing the timeout exponentially (whenever the view changes). We will refer to this parametrization by B. Finally, we propose another strategy, which consists of increasing the timeout exponentially every $t + 1$ views. In the context of leader-based algorithms, this strategy ensures that, if the timeout is large enough to terminate the started consensus instances, then a Byzantine leader will not be able to force correct processes to increase the timeout. We will refer to this last parameterizations by C. These three strategies are summarized in the following table, where v represents the view number and Γ_0 denotes the initial timeout.

Strategy	A	B	C
$\Gamma(v)$	$v\Gamma_0$	$2^{v-1}\Gamma_0$	$2^{\lfloor \frac{v-1}{t+1} \rfloor}\Gamma_0$

D. Correctness proofs of Algorithm 4

In the sequel, let τ_G denote the first time that the actual end-to-end transmission delay δ is reached. All messages sent before τ_G are received the latest by time $\tau_G + \delta$. Let v_0 denote the largest view number such that no correct process has sent a START message for view v_0 by time τ_G, but some correct process has sent a START message for view $v_0 - 1$. Let r_0 denote the largest round number such that no correct process has sent a START message for round r_0 by time τ_G, but some correct process has sent a START message for round $r_0 - 1$. We prove the results related to the view number, similar results hold for round numbers:

Lemma 1: Let p be a correct process that sends message $\langle \text{START}, -, v, -, p\rangle$ at some time τ_0, then at least one correct process q has sent message $\langle \text{INIT}, v, -, q\rangle$ at time $\tau \leq \tau_0$.

Proof: Assume by contradiction that no correct process q has sent message $\langle \text{INIT}, v, -, q\rangle$. This means that a correct process can receive at most t messages $\langle \text{INIT}, v, -, -\rangle$ in line 28. Therefore, no correct process executes line 32, and no correct process starts view v because of line 35, which is a contradiction. ■

Lemma 2: Let all correct processes p send message $\langle \text{INIT}, v, -, p\rangle$ at some time τ_0, then all correct processes p will send message $\langle \text{START}, -, v, -, p\rangle$ by time $max\{\tau_0, \tau_G\} + \delta$.

Proof: If all correct processes p send message $\langle \text{INIT}, v, -, p\rangle$ at some time τ_0, then all correct processes are in view $v-1$ at time τ_0 by lines 45-47. A correct process q in view $v - 1$, receives at least $n - t \geq 2t + 1$ messages $\langle \text{INIT}, v, -, p\rangle$ by time $\tau_0 + \delta$ if $\tau_0 \geq \tau_G$, or by time $\tau_G + \delta$ if $\tau_0 < \tau_G$. From lines 35 and 36, q starts view v by time $max\{\tau_0, \tau_G\} + \delta$. ■

Lemma 3: Every correct process p sends message $\langle \text{START}, -, v_0 - 1, -, p\rangle$ by time $\tau_G + 2\delta$.

Proof: We assume that there is a correct process p with $v_p = v_0 - 1$ at time τ_G. This means that p has received at least $2t + 1$ messages $\langle \text{INIT}, v_0 - 1, -, -\rangle$ (line 35). Or at least $t + 1$ correct processes are in view $v_0 - 2$ and have sent a message $\langle \text{INIT}, v_0 - 1, -, -\rangle$. These messages will be received by all correct processes the latest by time $\tau_G + \delta$. Therefore, all correct processes in view $< v_0 - 1$ receive at least $t + 1$ messages $\langle \text{INIT}, v_0 - 1, -, -\rangle$ by time $\tau_G + \delta$, start view $v_0 - 2$ (line 31) and send a message $\langle \text{INIT}, v_0 - 1, -, -\rangle$ (line 32). These messages are received by all correct processes by time $\tau_G + 2\delta$. Because $n - t > 2t$, all correct processes receive at least $2t+1$ messages $\langle \text{INIT}, v_0-1, -, -\rangle$ by time $\tau_G + 2\delta$ (line 35), start view $v_0 - 1$ (line 36), and send a message $\langle \text{START}, -, v_0 - 1, -, -\rangle$ (line 16). ■

Lemma 4: Let p be the first (not necessarily unique) correct process that sends message $\langle \text{START}, -, v, r, p\rangle$ with $v \geq v_0$ at some time $\tau \geq \tau_G$. Then no correct process sends message $\langle \text{START}, -, v+1, -, -\rangle$ before time $\tau+\Gamma(v)$. Moreover, no correct process sends message $\langle \text{INIT}, v + 2, -, -\rangle$ before time $\tau + \Gamma(v)$.

Proof: For the START message, assume by contradiction that process q is the first correct process that sends message $\langle \text{START}, -, v + 1, 1, q\rangle$ before time $\tau + \Gamma(v)$. Process q can send this message only if it receives $2t + 1$ messages $\langle \text{INIT}, v + 1, -, -\rangle$ (line 35), This means that at least $t + 1$ correct processes are in view v and have sent $\langle \text{INIT}, v + 1, -, -\rangle$. In order to send $\langle \text{INIT}, v + 1, -, -\rangle$, a correct process takes at least $\Gamma(v)$ time in view v (line 19). So message $\langle \text{START}, -, v+1, -, q\rangle$ is sent by correct process q at the earliest by time $\tau + \Gamma(v)$. A contradiction.

For the INIT message, since no correct process starts view $v+1$ before time $\tau+\Gamma(v)$, no correct process sends message $\langle \text{INIT}, v + 2, -, q\rangle$ before time $\tau + \Gamma(v)$. ■

Lemma 5: Let p be the first (not necessarily unique) correct process that sends message $\langle \text{START}, -, v, -, p\rangle$ with $v \geq v_0$ at some time $\tau \geq \tau_G$. Then every correct process q sends message $\langle \text{START}, -, v, -, q\rangle$ by time $\tau + 2\delta$.

Proof: Note that by the assumption, all view $v \geq v_0$ messages are sent at or after τ_G, and thus they are received by all correct processes δ time later. By Lemma 4, there is no message $\langle \text{START}, -, v', -, -\rangle$ with $v' > v$ in the system before $\tau + \Gamma(v)$. Process p sends message $\langle \text{START}, -, v, -, p\rangle$ if it receives $2t + 1$ messages $\langle \text{INIT}, v, -, -\rangle$ (line 35). This means that at least $t + 1$ correct processes are in view $v - 1$ and have sent message $\langle \text{INIT}, v, -, -\rangle$, the

latest by time τ. All correct processes in view $< v$ receive at least $t+1$ messages $\langle \text{INIT}, v, -, - \rangle$ the latest by time $\tau + \delta$, start view $v-1$ (line 31) and send $\langle \text{INIT}, v, -, - \rangle$ (line 32) which is received at most δ time later. Because $n - t > 2t$, every correct process q receives at least $2t+1$ messages $\langle \text{INIT}, v, -, - \rangle$ by time $\tau + 2\delta$ (line 35), start view v (line 36), and send message $\langle \text{START}, -, v, -, q \rangle$ (line 16). ■

Following two lemmas hold for round numbers.

Lemma 6: If a correct process p sends message $\langle \text{START}, -, v, r, p \rangle$ at time $\tau > \tau_G$, it will send message $\langle \text{START}, -, v, r+1, p \rangle$ the latest by time $\tau + 3\delta + \Gamma(v)$.

Proof: From Lemma 5 (similar result for round number), all correct processes q send message $\langle \text{START}, -, v, r, q \rangle$ the latest by time $\tau + 2\delta$. Then they wait for the timeout of round r which is $\Gamma(v)$ (lines 17 and 19). Therefore, by time $\tau + 2\delta + \Gamma(v)$ all correct processes timeout for round r, and send $\langle \text{INIT}, v, r+1, q \rangle$ message to all (line 20), which takes δ time to be received by all correct processes. Finally the latest by time $\tau + 3\delta + \Gamma(v)$, process p receives $n - t \geq 2t+1$ messages $\langle \text{INIT}, v, r+1, - \rangle$ and starts round $r+1$ (line 36). ■

Lemma 7: A timeout $\Gamma(v) \geq 3\delta$ for round r ensures that if a correct process p sends message $\langle \text{START}, -, v, r, p \rangle$ to all at time $\tau \geq \tau_G$, it will receive all round messages $\langle \text{START}, -, v, r, q \rangle$ from all correct processes q, before the expiration of the timeout (at time $\tau + 3\delta$).

Proof: From Lemma 5 (similar result for round number), all correct processes q send message $\langle \text{START}, -, v, r, q \rangle$ to all the latest by time $\tau + 2\delta$. The message of round r takes an additional δ time. Therefore a timeout of at least 3δ ensures the stated property. ■

Therefore, we have the following theorem:

Theorem 1: Algorithm 4 with $n > 3t$ ensures the existence of round r_0 such that $\forall r \geq r_0 : \mathcal{P}_{sync}(r)$.

Proof: The proof holds from the previous lemmas. ■

V. TIMING ANALYSIS

In this section we analyze the impact of the strategies A, B and C on our four consensus algorithms. We start with the analysis of the round implementation. Then we use these results to compute the execution time of k consecutive instances of consensus using the four algorithms MA-L, MA-D, CL-L and CL-D.

First, for each strategy A, B, C, we compute the best case and worst-case execution time of k instances of repeated consensus, based on two parameters α and β: The parameter α is the one used in Algorithm 4. It denotes the number of rounds per phase of an algorithm, i.e., the number of rounds needed to decide in the best case. Thus, α gives also the length of a view in case a process does not decide. The parameter β denotes the number of consecutive views in which a process might not decide although the timeout is already set to the correct value. This might happen when a faulty process is the leader.

	fault-free case		worst case	
	α	β	α	β
MA-D	$t+2$	0	$t+2$	0
MA-L	4	0	4	t
CL-D	$t+3$	0	$t+3$	0
CL-L	5	0	5	t

Table I
PARAMETERS FOR ALGORITHMS MA AND CL

A. Best case analysis

In the best case we have $\Gamma_0 = \delta$ and there are no faults. Processes start a round at the same time and a round takes 2δ (δ for the timeout and δ for the INIT messages), and processes decide at the end of each phase (=α rounds). Therefore, the decision for k consecutive instances of consensus occurs at time $2\delta\alpha k$. Obviously, the algorithm with the smallest α (that is, the leader-based or the decentralized with $t \leq 2$) performs in this case the best.

B. Worst case analysis

We compute now $\tau_X(k, \alpha, \beta)$, the worst-case execution time until the k^{th} decision when using the strategy $X \in \{A, B, C\}$. Based on item 3 in Section IV-B (and Lemma 7), the first decision does not occur until the round timeout is larger or equal to 3δ. We denote below with v_0 the view that corresponds to the first decision ($k = 1$).

Strategy A: With strategy A, the timeout is increased in each new view by Γ_0 until $v\Gamma_0 \geq 3\delta$, i.e., until $v = \lceil 3\delta/\Gamma_0 \rceil$. Then the timeout is increased for the next β views. Therefore, we have $v_0 = \lceil 3\delta/\Gamma_0 \rceil + \beta$. To compute the time until a decision, observe that a view v lasts $\Gamma(v)$ (timeout for view v) plus the time until all INIT messages are received. It can be shown that the latter takes at most 3δ (see item 2 in Section IV-B). Therefore we have for the worst case:

$$\tau_A(1, \alpha, \beta) = \sum_{v=1}^{v_0} \alpha(\Gamma(v) + 3\delta) = \alpha \sum_{v=1}^{v_0} (v\Gamma_0 + 3\delta) =$$
$$= \alpha \left(\frac{v_0(v_0+1)}{2} \Gamma_0 + 3\delta v_0 \right) =$$
$$= \alpha \left(\frac{\Gamma_0}{2} (\lceil 3\delta/\Gamma_0 \rceil + \beta)(\lceil 3\delta/\Gamma_0 \rceil + \beta + 1) + 3\delta(\lceil 3\delta/\Gamma_0 \rceil + \beta) \right) \quad (1)$$

and for $k > 1$,

$$\tau_A(k, \alpha, \beta) = \tau_A(k-1, \alpha, \beta) + \alpha(v_0\Gamma_0 + 3\delta) =$$
$$= \tau_A(k-1, \alpha, \beta) + \alpha(\lceil 3\delta/\Gamma_0 \rceil \Gamma_0 + \beta\Gamma_0 + 3\delta) \quad (2)$$

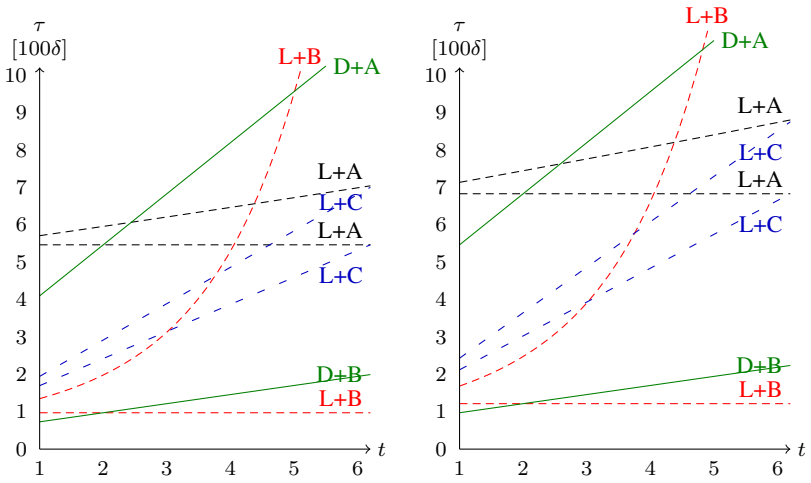

Figure 1. Comparison for $k = 1$. The lower curve represents the fault-free case and the higher curve represents the worst case.

Strategy B: With strategy B, the timeout doubles in each new view until $2^{v-1}\Gamma_0 \geq 3\delta$. In other words, the timeout doubles until reaching view $v = \left\lceil \log_2 \frac{6\delta}{\Gamma_0} \right\rceil$. Including β, we have $v_0 = \left\lceil \log_2 \frac{6\delta}{\Gamma_0} \right\rceil + \beta$, and:

$$\begin{aligned}
&\tau_B(1,\alpha,\beta) = \sum_{v=1}^{v_0} \alpha(\Gamma(v) + 3\delta) = \alpha \sum_{v=1}^{v_0} (2^{v-1}\Gamma_0 + 3\delta) = \\
&= \alpha\left((2^{v_0} - 1)\Gamma_0 + 3\delta v_0\right) = \\
&= \alpha\left(\left(2^{\left\lceil \log_2 \frac{6\delta}{\Gamma_0} \right\rceil + \beta} - 1\right)\Gamma_0 + 3\delta\left(\left\lceil \log_2 \frac{6\delta}{\Gamma_0} \right\rceil + \beta\right)\right) = \\
&= \alpha\left(2^{\left\lceil \log_2 \frac{3\delta}{\Gamma_0} \right\rceil} 2^{\beta+1}\Gamma_0 - \Gamma_0 + 3\delta\left\lceil \log_2 \frac{3\delta}{\Gamma_0} \right\rceil + 3\delta + 3\delta\beta\right)
\end{aligned} \tag{3}$$

and for $k > 1$,

$$\begin{aligned}
\tau_B(k,\alpha,\beta) &= \tau_B(k-1,\alpha,\beta) + \alpha(2^{v_0-1}\Gamma_0 + 3\delta) = \\
&= \tau_B(k-1,\alpha,\beta) + \alpha\left(2^{\left\lceil \log_2 \frac{3\delta}{\Gamma_0} \right\rceil} 2^{\beta}\Gamma_0 + 3\delta\right)
\end{aligned} \tag{4}$$

Strategy C: Finally, for strategy C, the timeout doubles in each new view until $2^{\frac{v-1}{t+1}}\Gamma_0 \geq 3\delta$. In other words, the timeout doubles until reaching view $v = 1 + (t+1)\left\lceil \log_2 \frac{3\delta}{\Gamma_0} \right\rceil$; then it remains the same for the next β views. Therefore we have $v_0 = (t+1)\left\lceil \log_2 \frac{3\delta}{\Gamma_0} \right\rceil + \beta + 1$, and:[4]

[4]Note that from $v = 1 + (t+1)\left\lceil \log_2 \frac{3\delta}{\Gamma_0} \right\rceil$ it follows that $\frac{v-1}{t+1}$ is an integer.

$$\begin{aligned}
&\tau_C(1,\alpha,\beta) = \\
&= \alpha\left((t+1)\sum_{l=0}^{\frac{v-1}{t+1}-1}(2^l\Gamma_0 + 3\delta) + (\beta+1)\left(2^{\frac{v-1}{t+1}}\Gamma_0 + 3\delta\right)\right) = \\
&= \alpha(t+1)\left(2^{\frac{v-1}{t+1}}\Gamma_0 - \Gamma_0 + 3\delta\frac{v-1}{t+1}\right) \\
&+ \alpha(\beta+1)\left(2^{\frac{v-1}{t+1}}\Gamma_0 + 3\delta\right) = \\
&= \alpha(t+1)\left(2^{\left\lceil \log_2 \frac{3\delta}{\Gamma_0} \right\rceil}\Gamma_0 - \Gamma_0 + 3\delta\left\lceil \log_2 \frac{3\delta}{\Gamma_0} \right\rceil\right) \\
&+ \alpha(\beta+1)\left(2^{\left\lceil \log_2 \frac{3\delta}{\Gamma_0} \right\rceil}\Gamma_0 + 3\delta\right)
\end{aligned} \tag{5}$$

and for $k > 1$,

$$\begin{aligned}
\tau_C(k,\alpha,\beta) &= \tau_C(k-1,\alpha,\beta) + \alpha\left(2^{\frac{v-1}{t+1}}\Gamma_0 + 3\delta\right)(\beta+1) = \\
&= \tau_C(k-1,\alpha,\beta) + \alpha\left(2^{\left\lceil \log_2 \frac{3\delta}{\Gamma_0} \right\rceil}\Gamma_0 + 3\delta\right)(\beta+1)
\end{aligned} \tag{6}$$

Note that strategy C makes sense only for leader-based algorithms.

1) Comparison: Table I gives α and β for all algorithms we discussed. For the worst case analysis we distinguish two cases: the worst *fault-free case*, which is the worst case in terms of the timing for a run without faulty process; and the general *worst-case* that gives the values for a run in which t processes are faulty.

We compare our results graphically in Figures 1-3. The execution time for each algorithm and strategy is a function of k, t, and the ratio δ/Γ_0. In the sequel, we fix two of these variables and vary the third.

We first focus on the first instance of consensus, that is, we fix $k = 1$ and assume $\delta = 10\Gamma_0$ which gives

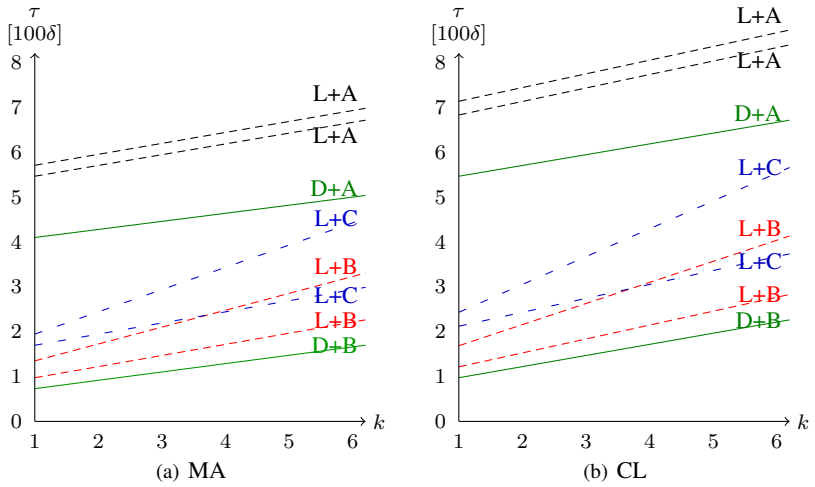

Figure 2. Comparison for $t = 1$. The lower curve represents the fault-free case and the higher curve represents the worst case.

$\lceil \log_2(3\delta/\Gamma_0) \rceil = 5$, i.e., the transmission delay is estimated correctly after five times doubling the timeout. The result is depicted in Fig. 1. We first observe, as expected, that the fault-free case and the worst-case are the same for the decentralized versions. For the—in real systems relevant—cases $t < 3$, for each strategy, the decentralized algorithm decides even faster in the worst-case than the leader-based version of the same algorithm in the fault-free case. For larger t, the leader-based algorithms with strategy B are faster in the fault-free case, but less performant in the worst-case.

Next, we look how the algorithms perform for multiple instances of consensus. To this end, we depict the total time until k consecutive instances decide in Fig. 2, for the most relevant case $t = 1$. Again we assume $\delta = 10\Gamma_0$. Here, the decentralized algorithm is always superior to the leader-based variant using the same strategy, in the sense that even in the worst case it is faster than the corresponding algorithm in the best case. In absolute terms, the decentralized algorithms with strategy B perform the best.

Finally, we analyze the impact of the choice of Γ_0 on the execution time (Fig. 3). This is relevant only for the first decision, i.e., $k = 1$. We look at the case $t = 1$ and vary $\log_2 \frac{3\delta}{\Gamma_0}$. Again, the decentralized version is superior for each strategy. However, it can be seen that strategy A is not a good choice, neither with a decentralized nor with a leader-based algorithm, if $\log_2 \frac{3\delta}{\Gamma_0}$ is too large.

VI. Conclusion

We compared the leader-based and the decentralized variant of two typical consensus algorithms for Byzantine faults in an analytical way.

The results show a surprisingly clear preference for the decentralized version. While always having a better worst-cast performance, for the practically relevant cases $t \leq 2$, the decentralized variant of the algorithm is at least as good as even the fault-free case scenarios of the leader-based algorithms. But also in the best case, for $t \leq 2$, the decentralized solution is at least as good as the leader-based variant.

References

[1] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, 1980.

[2] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[3] M. Ben-Or, "Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols," in *PODC'83*. NY, USA: ACM, 1983, pp. 27–30.

[4] M. Rabin, "Randomized Byzantine generals," in *Proc. Symposium on Foundations of Computer Science*, 1983, pp. 403–409.

[5] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *JACM*, vol. 35, no. 2, pp. 288–323, apr 1988.

[6] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *JACM*, vol. 43, no. 2, pp. 225–267, Mar. 1996.

[7] L. Lamport, "The part-time parliament," *ACMTCS*, vol. 16, no. 2, pp. 133–169, May 1998.

[8] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[9] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *DSN'08*, 2008, pp. 197–206.

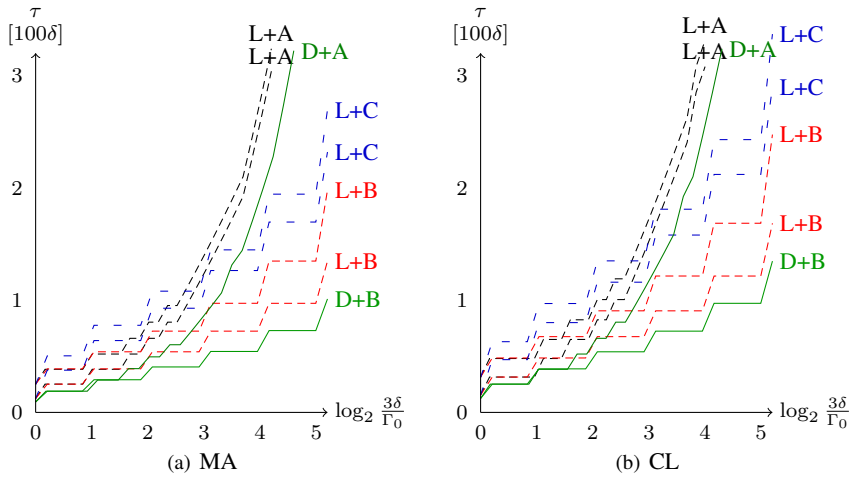

Figure 3. Comparison of different strategies with $k = 1$ and $t = 1$. The lower curve represents the fault-free case and the higher curve represents the worst case.

[10] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine fault tolerant systems tolerate Byzantine faults," in *NSDI'09*. Berkeley, CA, USA: USENIX Association, 2009, pp. 153–168.

[11] F. Borran and A. Schiper, "A Leader-Free Byzantine Consensus Algorithm," in *ICDCN*, ser. Lecture Notes in Computer Science (LNCS). Berlin: Springer-Verlag, 2010, pp. 67–78. [Online]. Available: http://www.icdcn.org/

[12] J.-P. Martin and L. Alvisi, "Fast Byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, jul 2006. [Online]. Available: http://www.cs.utexas.edu/users/lasr/papers/Martin06Fast.pdf

[13] Z. Milosevic, M. Hutle, and A. Schiper, "Unifying byzantine consensus algorithms with weak interactive consistency," in *OPODIS*, 2009, pp. 300–314.

[14] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 45–58, 2007.

[15] T. K. Srikanth and S. Toueg, "Optimal clock synchronization," *J. ACM*, vol. 34, no. 3, pp. 626–645, 1987.

[16] M. Hutle and A. Schiper, "Communication predicates: A high-level abstraction for coping with transient and dynamic faults," in *Dependable Systems and Networks (DSN 2007)*. IEEE, Jun. 2007, pp. 92–10.